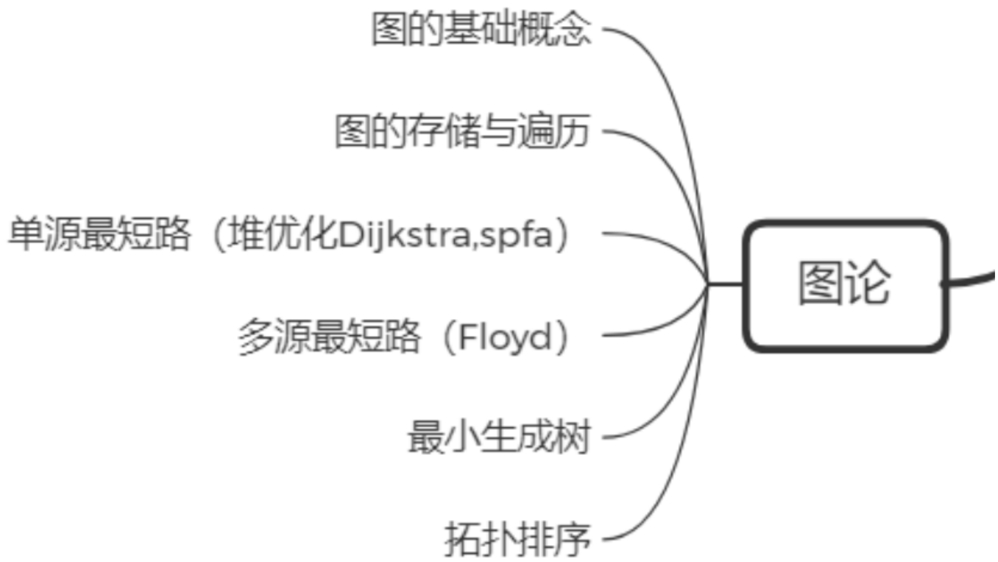
蓝桥杯十天冲刺省一


图的基础概念
图的存储与遍历
单源最短路（堆优化Dijkstra,spfa）
多源最短路（Floyd）
最小生成树
拓扑排序
图论

Day-8 图

图的种类

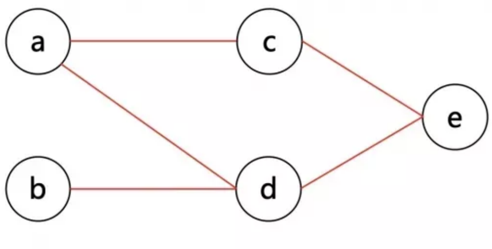
图有多种，包括 **无向图 (Undirected graph)** ， **有向图 (Directed graph)** 等

若 G 为无向图，则 E 中的每个元素为一个无序二元组 (u, v) ，称作 **无向边 (Undirected edge)** ，简称 **边 (Edge)** ，其中 $u, v \in V$ 。设 $e = (u, v)$ ，则 u 和 v 称为 e 的 **端点 (Endpoint)** 。

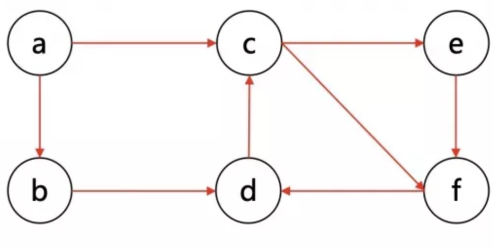
若 G 为混合图，则 E 中既有向边，又有无向边。

若 G 的每条边 $e_k = (u_k, v_k)$ 都被赋予一个数作为该边的 **权** ，则称 G 为 **赋权图** 。如果这些权都是正实数，就称 G 为 **正权图** 。

形象地说，图是由若干点以及连接点与点的边构成的。



无向图



有向图

度数

与一个顶点 v 关联的边的条数称作该顶点的 **度 (Degree)** ，记作 $d(v)$ 。特别地，对于边 (v, v) ，则每条这样的边要对 $d(v)$ 产生 2 的贡献。

对于无向简单图，有 $d(v) = |N(v)|$ 。

握手定理（又称图论基本定理）：对于任何无向图 $G = (V, E)$ ，有 $\sum_{v \in V} d(v) = 2|E|$ ，即无向图中结点度数的总和等于边数的两倍。有向图中结点的入度之和等于出度之和等于边数。

推论： **在任意图中，度数为奇数的点必然有偶数个。**

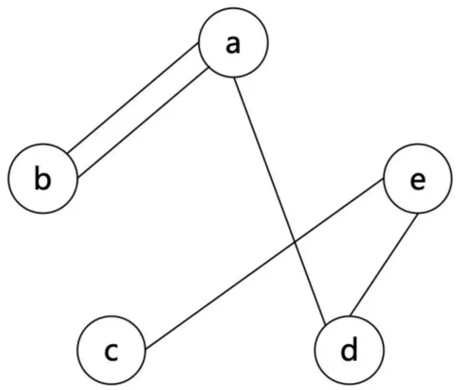
简单图

自环 (Loop)： 对 E 中的边 $e=(u,v)$，若 $u=v$，则 e 被称作一个自环。

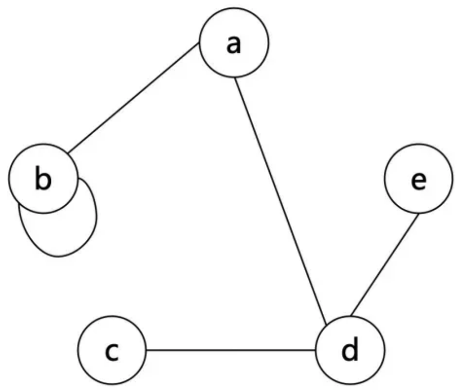
重边/平行边 (Multiple edge)： 若 E 中存在两个完全相同的元素（边）e_1,e_2，则它们被称作（一组）重边。

简单图 (Simple graph)： 若一个图中 **没有自环和重边**，它被称为简单图。非空简单无向图中一定存在度相同的结点。

如果一张图中有自环或重边，则称它为 **多重图 (Multigraph)** 。



平行边



自环边

在无向图中 (u,v) 和 (v,u) 算一组重边，而在有向图中， $u\to v$ 和 $v\to u$ 不为重边。

在题目中，如果没有特殊说明，是可以存在自环和重边的，在做题时需特殊考虑。

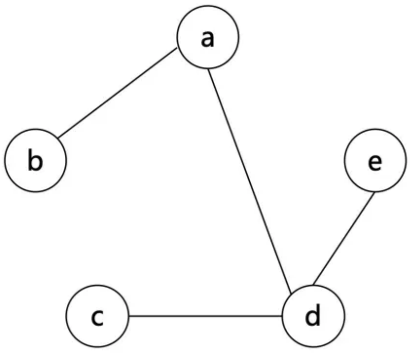
连通

无向图

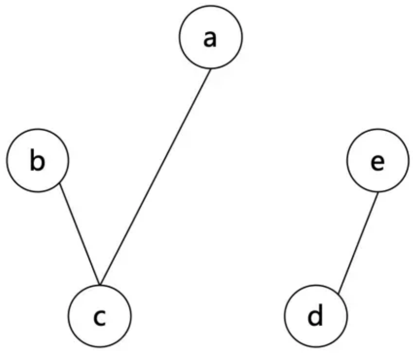
对于一张无向图 $G=(V,E)$，对于 $u,v\in V$，若存在一条途径使得 $v_0=u,v_k=v$，则称 u 和 v 是 **连通的 (Connected)** 。由定义，任意一个顶点和自身连通，任意一条边的两个端点连通。

若无向图 $G=(V,E)$ ，满足其中任意两个顶点均连通，则称 G 是 **连通图 (Connected graph)** ， G 的这一性质称作 **连通性 (Connectivity)** 。

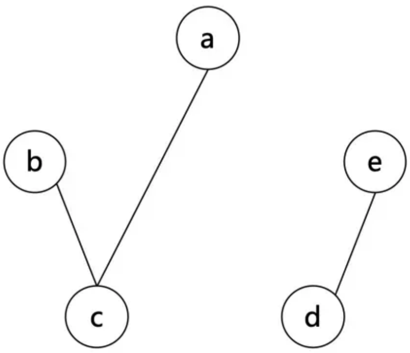
若 H 是 G 的一个连通子图，且不存在 F 满足 $H \subsetneq F \subseteq G$ 且 F 为连通图，则 H 是 G 的一个 **连通块/连通分量 (Connected component)** （极大连通子图）。



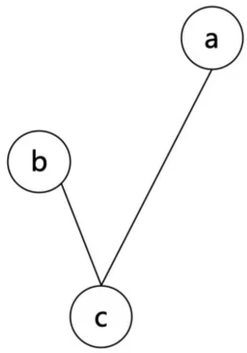
连通图



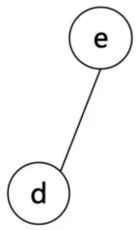
非连通图



图G



连通分量

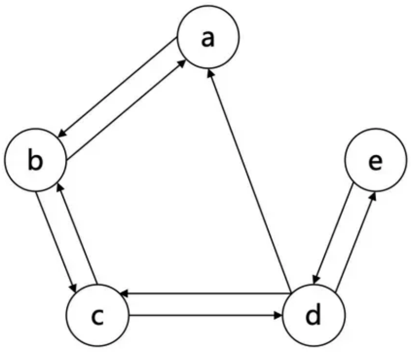


连通分量

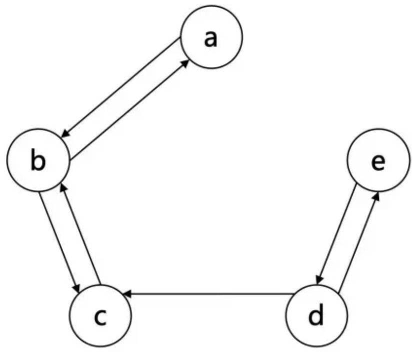
有向图

对于一张有向图 $G=(V,E)$ ，对于 $u,v \in V$ ，若存在一条途径使得 $v_0=u, v_k=v$ ，则称 u **可达** v 。由定义，任意一个顶点可达自身，任意一条边的起点可达终点。（无向图中的连通也可以视作双向可达。）

若一张有向图的节点两两互相可达，则称这张图是 **强连通的 (Strongly connected)** 。



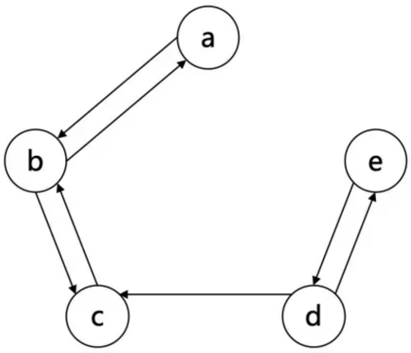
强连通图



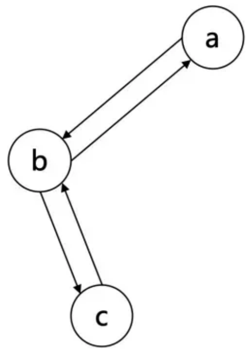
非强连通图

若一张有向图的边替换为无向边后可以得到一张连通图，则称原来这张有向图是 **弱连通的 (Weakly connected)** 。

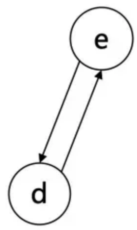
与连通分量类似，也有 **弱连通分量 (Weakly connected component)** （极大弱连通子图）和 **强连通分量 (Strongly Connected component)** （极大强连通子图）。



图G

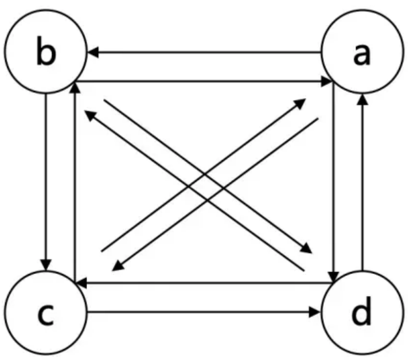


强连通分量

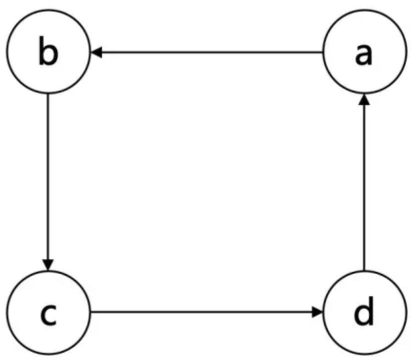


强连通分量

n 个顶点的强连通图最多 $n(n-1)$ 条边，最少 n 条边。



最多n(n-1)



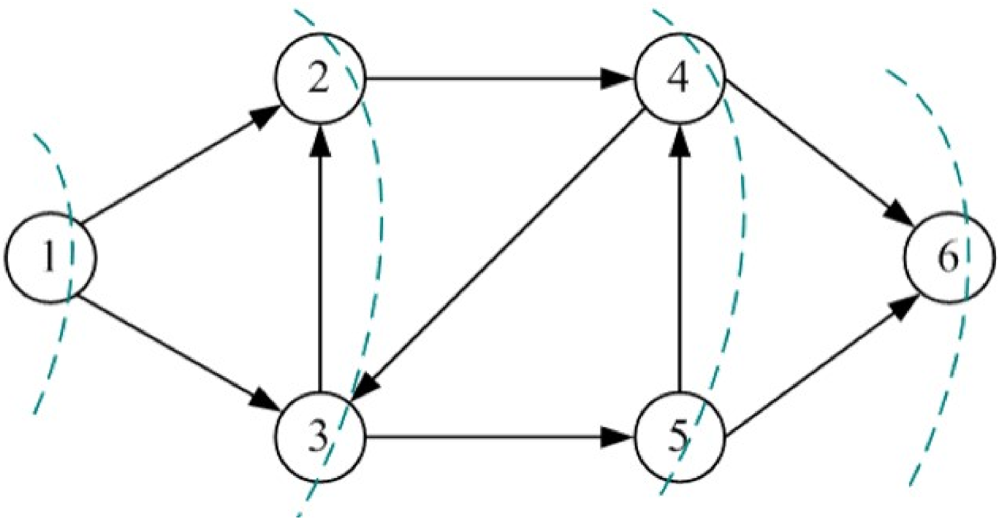
最少n

广度优先搜索

广度优先搜索 (Breadth First Search，BFS)，又称为宽度优先搜索，是最常见的图搜索方法之一

广度优先搜索是从某个顶点(源点)出发，一次性访问所有未被访问的邻接点，再依次从这些访问过的邻接点出发，……，似水中涟漪，似声音传播，一层一层地传播开来

广度优先遍历是按照**广度优先搜索**的方式对图进行遍历



广度优先遍历

广度优先遍历秘籍： 先被访问的顶点，其邻接点先被访问

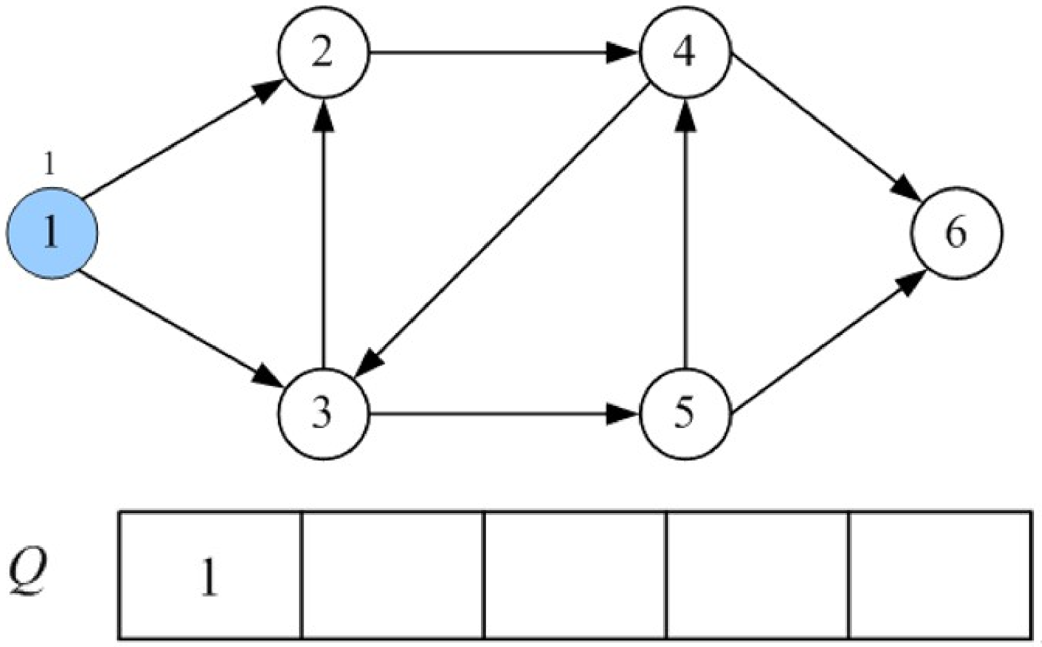
根据广度优先遍历，先来先服务，可以借助于**队列**实现。**每个节点访问一次且仅被访问一次**。又因为**图中若有环，某个顶点可能会被重复访问**，因此需要设置一个**辅助数组**

```
vis[i] = 0;//表示第i个顶点未访问
vis[i] = 1;//表示第i个顶点已访问
```

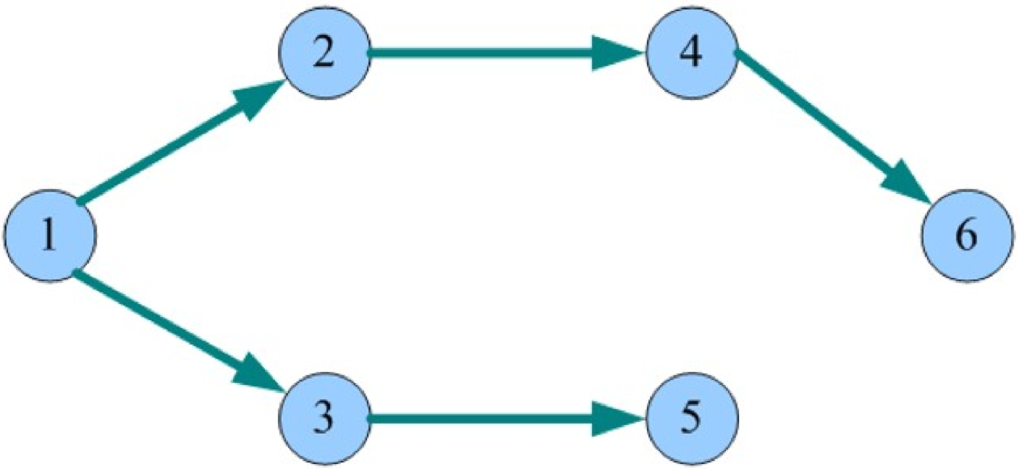
算法步骤

1. 初始化图中所有顶点未被访问，初始化一个空队列
2. 从图中某个顶点 u 出发，访问 u 并标记已访问，将 u 入队列

3. 如果队列非空，则继续执行，否则算法结束
4. 队头元素 v 出队列，依次访问 v 的所有未被访问的邻接点，标记已访问并入队列。转向步骤 3



如果把经过的路径画出来的话，会得到一棵**广度优先搜索树**



基于邻接矩阵的广度优先遍历

图的邻接矩阵存储方法需要特别注意图中顶点数，很容易爆空间

```
#include<bits/stdc++.h>
using namespace std;


const int N = 1e3 + 10;

```

```cpp
bool a[N][N]; // 邻接矩阵表示图的连接关系
bool vis[N];  // 标记顶点是否被访问过
int n, m, x, y; // n为顶点数，m为边数，x和y为边的两个端点

/**
 * @brief 进行广度优先搜索 (BFS)
 * @param u 起始顶点
 */
void bfs(int u) {
    queue<int> q; // 使用队列保存待访问的顶点
    vis[u] = true; // 标记起始顶点为已访问
    q.push(u); // 将起始顶点入队

    while (!q.empty()) { // 当队列不为空时
        int k = q.front(); // 取出队列头部顶点
        q.pop(); // 将队列头部顶点出队

        for (int i = 1; i <= n; i++) { // 遍历顶点k的所有邻接点
            if (a[k][i] && !vis[i]) { // 如果顶点k和顶点i相邻且顶
点i未被访问过
                vis[i] = true; // 标记顶点i为已访问
                q.push(i); // 将顶点i入队
            }
        }
    }
}

int main() {
    cin >> n >> m; // 输入顶点数和边数

    // 读入边的信息并构建邻接矩阵
    for (int i = 1; i <= m; i++) {
        cin >> x >> y;
        a[x][y] = true;
        // 如果是无向图，则需要同时标记a[y][x]
    }

    bfs(1); // 从顶点1开始进行广度优先搜索
```

```
    return 0;
}
```

基于邻接表的广度优先遍历

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 10;

vector<int> a[N]; // 图的邻接表表示法
bool vis[N]; // 标记顶点是否访问过
int n, m, x, y; // 图中n个顶点，m条边，以及边的两个端点

/**
 * @brief 进行广度优先搜索 (BFS)
 * @param u 起始顶点
 */
void bfs(int u) {
    queue<int> q; // 创建一个普通队列（先进先出）
    vis[u] = true; // 标记源点为已访问
    q.push(u); // 源点入队列

    while (!q.empty()) { // 当队列不为空时
        int k = q.front(); // 取出队头元素
        q.pop(); // 队头元素出队

        for (auto v : a[k]) { // 遍历顶点k的所有邻接点
            if (!vis[v]) { // 如果顶点v未被访问
                vis[v] = true; // 标记顶点v为已访问
                q.push(v); // 将顶点v入队
            }
        }
    }
}

int main() {
    cin >> n >> m; // 输入顶点数和边数
```

```
    // 读入边的信息并构建邻接表
    for (int i = 1; i <= m; i++) {
        cin >> x >> y;
        a[x].push_back(y);
        // 如果是无向图，则需要同时添加a[y].push_back(x);
    }

    bfs(1); // 从顶点1开始进行广度优先搜索

    return 0;
}
```

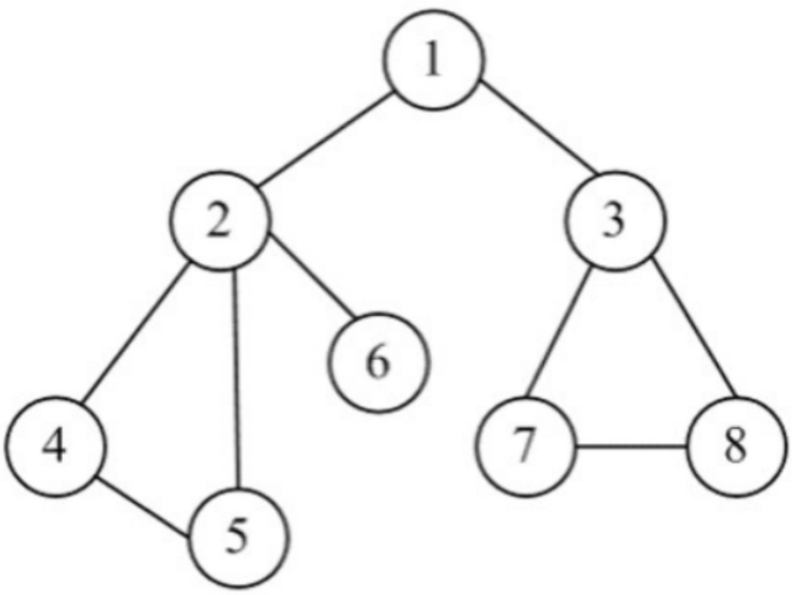
[迷宫（蓝桥杯C/C++2019B组省赛）](#)

深度优先搜索

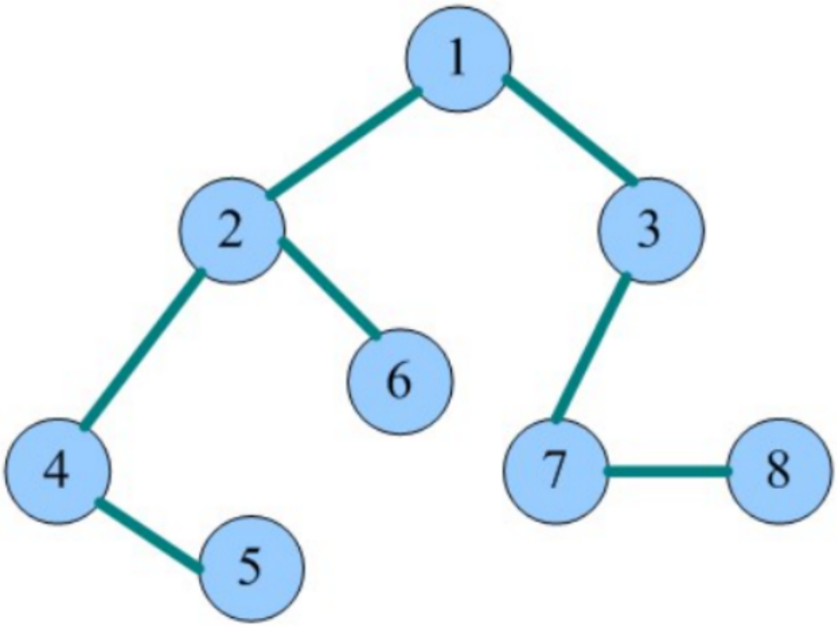
深度优先搜索(Depth First Search，DFS)，是最常见的图搜索方法之一

深度优先搜索沿着一条路径一直走下去，无法行进时，回退到刚刚访问的顶点，似不撞南墙不回头，不到黄河不死心

深度优先遍历是按照**深度优先搜索**的方式对图进行遍历



如果把经过的路径画出来的话，会得到一棵**深度优先搜索树**



深度优先遍历

深度优先遍历秘籍：后被访问的顶点，其邻接点先被访问

根据深度优先遍历秘籍，后来先服务，可以借助于**栈**实现。**递归本身就是使用栈实现的**，因此**使用递归更方便**

每个顶点访问且只访问一次，又因为**图中若有环，某个顶点可能会被重复访问**，因此需要设置一个**辅助数组**

```
vis[i] = 0;//表示第i个顶点未访问
vis[i] = 1;//表示第i个顶点已访问
```

算法步骤

1. 初始化图中所有顶点未被访问
2. 从图中某个顶点 u 出发，访问 u 并标记已访问
3. 依次检查 u 的所有邻接点 v，如果 v 未被访问，则从 v 出发深度优先遍历

基于邻接矩阵的深度优先遍历

图的邻接矩阵存储方法需要特别注意图中顶点数，很容易爆空间

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e3 + 10;

// 边上不带权值的图的邻接矩阵
// 如果带权图，邻接矩阵需要定义为int
bool a[N][N]; // 邻接矩阵表示法需要关注图中顶点数，避免爆空间
bool vis[N]; // 标记是否访问过
int n, m, x, y;

/**
 * @brief 进行深度优先搜索 (DFS)
 * @param u 起始顶点
 */
void dfs(int u) {
    vis[u] = true; // 标记顶点u为已访问
    cout << u << " "; // 输出当前顶点u
    for (int i = 1; i <= n; i++) { // 依次检查顶点u的所有邻接点
        if (a[u][i] && !vis[i]) { // 如果顶点u和顶点i邻接且顶点i
未被访问
            dfs(i); // 从顶点i开始递归深度优先遍历
        }
    }
}

int main() {
    cin >> n >> m; // 输入图中顶点数n和边数m
    for (int i = 1; i <= m; i++) {
        cin >> x >> y;
        a[x][y] = true;
        // 如果是无向图，则需要同时标记a[y][x]为true
    }
    dfs(1); // 从顶点1开始进行深度优先搜索
    return 0;
```

```
}
```

基于邻接表的深度优先遍历

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 10;

vector<int> a[N]; // 图的邻接表表示法
bool vis[N]; // 标记顶点是否访问过
int n, m, x, y;

/**
 * @brief 进行深度优先搜索 (DFS)
 * @param u 起始顶点
 */
void dfs(int u) {
    vis[u] = true; // 标记顶点u为已访问
    cout << u << " "; // 输出当前顶点u
    for (auto v : a[u]) { // 依次检查顶点u的所有邻接点
        if (!vis[v]) { // 如果顶点v未被访问
            dfs(v); // 从顶点v开始递归深度优先遍历
        }
    }
}

int main() {
    cin >> n >> m; // 输入图中顶点数n和边数m
    for (int i = 1; i <= m; i++) {
        cin >> x >> y;
        a[x].push_back(y);
        // 如果是无向图，则需要同时添加 a[y].push_back(x);
    }
    dfs(1); // 从顶点1开始进行深度优先搜索
    return 0;
}
```

[回忆迷宫（蓝桥杯Java2022B组省赛）]

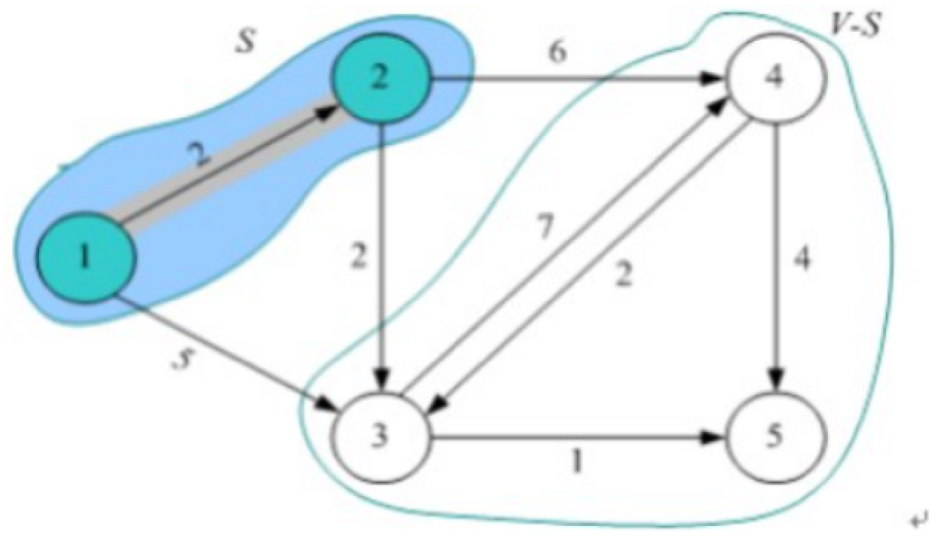
朴素Dijkstra算法

算法思想

$Dijkstra$ 算法是解决**单源最短路径问题**的**贪心算法**，**它先求出长度最短的一条路径，再参照该最短路径求出长度次短的一条路径，直到求出源点到其他各个节点的最短路径**

$Dijkstra$ 算法基本思想：将顶点集合 V 划分为**两部分**：**集合** S 和 **集合** $V-S$，**其中** S **中的顶点到源点的最短路径已经确定，** $V-S$ **中的节点到源点的最短路径待定。**

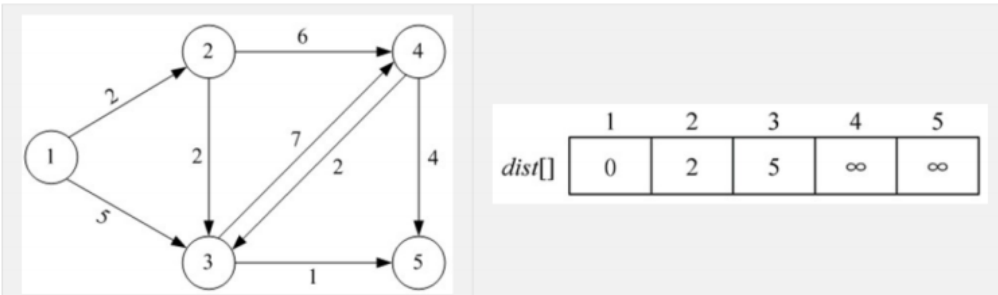
从源点出发只经过 S **中的节点到达 V−S 中的顶点的路径称为特殊路径**。$Dijkstra$ 算法的**贪心策略**是选择最短的特殊路径长度 $dist[t]$，并将顶点 t 加入到集合 S 中，同时借助 t 更新数组 $dist[]$。一旦 S 包含了所有顶点，$dist[]$ **就是从源点到其他节点的最短路径长度**



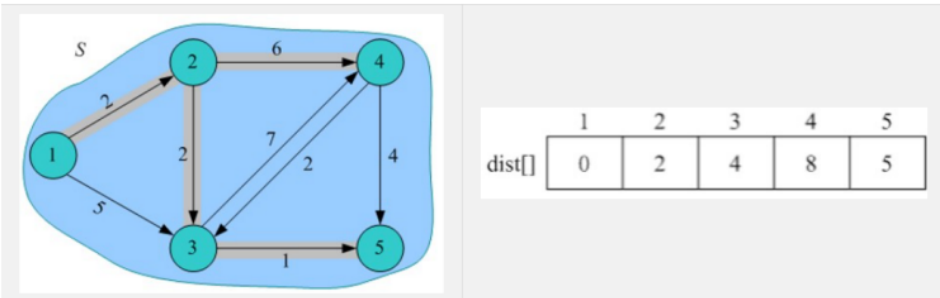
算法设计

1. **数据结构**：**邻接矩阵** $G[][]$ **存储图**，$dist[i]$ 记录从源点到节点 i 的最短路径长度，如果 $flag[i]$ 等于 $true$，说明节点 i 已加入集合 S，否则 i 属于集合 $V-S$。
2. **初始化**：假设 u 为源点，令集合 $S=u$，对 $V-S$ 集合中的节点 i，初始化 $dist[i]=G[u][i]$
3. **找最小**：按照**贪心策略**来查找 $V-S$ 集合中 $dist[]$ 最小的顶点 t ，t 就是 $V-S$ 集合中距离源点 u 最近的顶点。将顶点 t 加入集合 S 中
4. **松弛操作**：对 $V-S$ 集合中所有节点 j，考察是否可以借助 t 得到**更短的路径**。如果源点 u 经过 t 到 j 的路径更短，即 $dist[j]>dist[t]+G[t][j]$ ，则更新 $dist[j]=dist[t]+G[t][j]$，即**松弛操作**
5. 重复执行**步骤** 3 和**步骤** 4 ，直到 $V-S$ 为空

[例题]求源点1到其它各个顶点的最短路径



过程同学们自己模拟上述算法设计步骤计算，最终结果为



堆优化版本Dijkstra算法

朴素版Dijkstar算法找最小值方法

```
int temp=INF,t;
for(int j=1;j<=n;j++)//在集合V-S中寻找距离源点u最近的顶点t
{
    if(flag[j]==0&&dist[j]<temp)
    {
        t=j;
        temp=dist[j];
    }
}
```

按照 **贪心策略**查找 $V-S$ 集合中 $dist[]$ 最小的顶点 t，其时间复杂度为 $O(n)$，如果使用**优先队列**，则每次找最小值时间复杂度降为 $O(logn)$，找最小值的总时间复杂度为 $O(nlogn)$

邻接矩阵写法

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1005;
const int INF = 0x3f3f3f3f; // 无穷大
int g[N][N], dist[N]; // g[][]为邻接矩阵，dist[i]表示源点到结点i的最短路径长度
int n, m; // n为顶点数，m为边数
bool flag[N]; // 如果flag[i]等于true，说明结点i已经加入到S集合；否则i属于V-S集合

struct node {
    int v, dis; // 顶点v，源点到v的最短路径长度dis
    bool operator < (const node &a) const { // 重载<，优先队列优先级，dis越小越优先
        return dis > a.dis;
    }
};
```

```
void dijkstra(int u) {
    priority_queue<node> q; // 优先队列优化
    memset(dist, 0x3f, sizeof(dist));
    dist[u] = 0;
    q.push({u, 0});
    while (!q.empty()) {
        node it = q.top(); // 优先队列队头元素为dist最小值
        q.pop();
        int t = it.v;
        if (flag[t]) // 说明已经找到了最短距离，该顶点是队列里面的重复元素
            continue;
        flag[t] = true;
        for (int j = 1; j <= n; j++) { // 松弛操作
            if (!flag[j] && dist[j] > dist[t] + g[t][j]) {
                dist[j] = dist[t] + g[t][j];
                q.push({j, dist[j]}); // 把更新后的最短距离压入优先队列，注意：里面的元素有重复
            }
        }
    }
}

int main() {
    int u, v, w, st; // u,v表示顶点，w表示u--v的距离,st表示源点
    cin >> n >> m;
    memset(g, 0x3f, sizeof(g));
    while (m--) {
        cin >> u >> v >> w;
        g[u][v] = min(g[u][v], w); // 邻接矩阵储存，保留最小的距离
    }
    // cin >> st; // 输入源点
    st = 1;
    dijkstra(st);
    if (dist[n] == INF)
        cout << -1;
    else
        cout << dist[n];
```

```
    return 0;
}
```

邻接表写法

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1005; // 顶点数是多少具体看题目
const int INF = 0x3f3f3f3f; // 无穷大
int dist[N]; // dist[i]表示源点到结点i的最短路径长度
int n, m; // n为顶点数，m为边数
bool flag[N]; // 如果flag[i]等于true,说明结点i已经加入到S集合;否则i属于V-S集合

struct node {
    int v, w; // 到v的路径长度为w
    bool operator < (const node &a) const { // 重载<,优先队列优先级,dis越小越优先
        return w > a.w;
    }
};

vector<node> g[N];

void dijkstra(int u) {
    priority_queue<node> q; // 优先队列优化
    memset(dist, 0x3f, sizeof(dist));
    dist[u] = 0;
    q.push({u, 0});
    while (!q.empty()) {
        node it = q.top(); // 优先队列队头元素为dist最小值
        q.pop();
        int t = it.v;
        if (flag[t]) // 说明已经找到了最短距离，该结点是队列里面的重复元素
            continue;
        flag[t] = true;
```

```
        for (auto e : g[t]) // 松弛操作
        {
            int v = e.v, w = e.w;
            if (dist[v] > dist[t] + w) {
                dist[v] = dist[t] + w;
                q.push({v, dist[v]}); // 把更新后的最短距离压入优先队列，注意：里面的元素有重复
            }
        }
    }
}

int main() {
    int u, v, w, st; // u,v表示顶点,w表示u--v的距离,st表示源点
    cin >> n >> m;
    while (m--) {
        cin >> u >> v >> w;
        g[u].push_back({v, w});
    }
    // cin >> st; // 输入源点
    st = 1;
    dijkstra(st);
    if (dist[n] == INF)
        cout << -1;
    else
        cout << dist[n];
    return 0;
}
```

练习题

路径（蓝桥杯C/C++2021B组省赛）

奶牛跨栏

最小生成树

生成树

对连通图进行遍历，过程中所经过的边和顶点的组合可看做是一棵普通树，通常称为生成树。

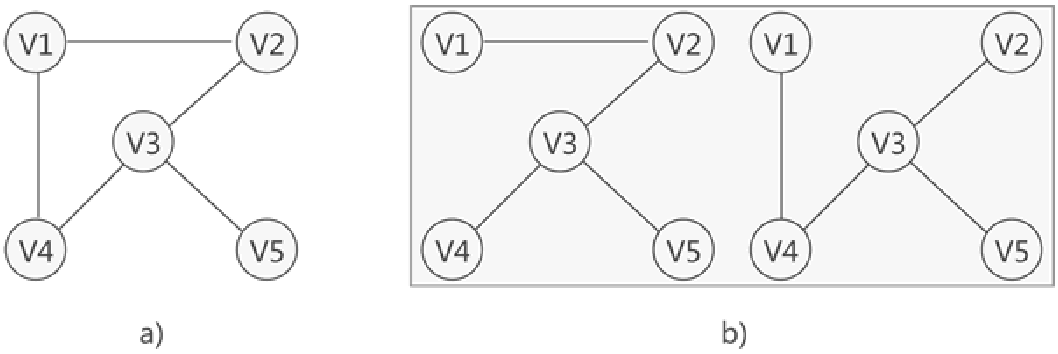


图 1 连通图及其对应的生成树

如图 1 所示，图 1a) 是一张连通图，图 1b) 是其对应的 2 种生成树。

连通图中，由于任意两顶点之间可能含有多条通路，遍历连通图的方式有多种，往往一张连通图可能有多种不同的生成树与之对应。

连通图中的生成树必须满足以下 2 个条件：

1. 包含连通图中所有的顶点；
2. 任意两顶点之间有且仅有一条通路；

因此，连通图的生成树具有这样的特征，即生成树中 `边的数量 = 顶点数 - 1` 。

最小生成树

我们定义无向连通图的 **最小生成树**（Minimum Spanning Tree，MST）为边权和最小的生成树。

注意：只有连通图才有生成树。

克鲁斯卡尔算法(Kruskal算法)

适用于稀疏图，时间复杂度 $O(mlogm)$。

核心思想：从小到大挑不多余的边，属于贪心的算法。

之前介绍了求最小生成树之普里姆算法。该算法从顶点的角度为出发点，时间复杂度为 $O(n^2)$，更适合与解决边的绸密度更高的连通网。

本节所介绍的克鲁斯卡尔算法，从边的角度求网的最小生成树，时间复杂度为 $O(E\log E)$。和普里姆算法恰恰相反，更适合于求边稀疏的网的最小生成树。

对于任意一个连通网的最小生成树来说，在要求总的权值最小的情况下，最直接的想法就是将连通网中的所有边按照权值大小进行升序排序，从小到大依次选择。

由于最小生成树本身是一棵生成树，所以需要时刻满足以下两点：

- 生成树中任意顶点之间有且仅有一条通路，也就是说，生成树中不能存在回路；
- 对于具有 n 个顶点的连通网，其生成树中只能有 $n-1$ 条边，这 $n-1$ 条边连通着 n 个顶点。

连接 n 个顶点在不产生回路的情况下，只需要 $n-1$ 条边。

思路

所以克鲁斯卡尔算法的具体思路是：将所有边按照权值的大小进行升序排序，然后从小到大一一判断，条件为：如果这个边不会与之前选择的所有边组成回路，就可以作为最小生成树的一部分；反之，舍去。直到具有 n 个顶点的连通网筛选出来 $n-1$ 条边为止。筛选出来的边和所有的顶点构成此连通网的最小生成树。

判断是否会产生回路的方法为：在初始状态下给每个顶点赋予不同的标记，对于遍历过程的每条边，其都有两个顶点，判断这两个顶点的标记是否一致，如果一致，说明它们本身就处在一棵树中，如果继续连接就会产生回路；如果不一致，说明它们之间还没有任何关系，可以连接。

过程

假设遍历到一条由顶点 A 和 B 构成的边，而顶点 A 和顶点 B 标记不同，此时不仅需要将顶点 A 的标记更新为顶点 B 的标记，还需要更改所有和顶点 A 标记相同的顶点的标记，全部改为顶点 B 的标记。

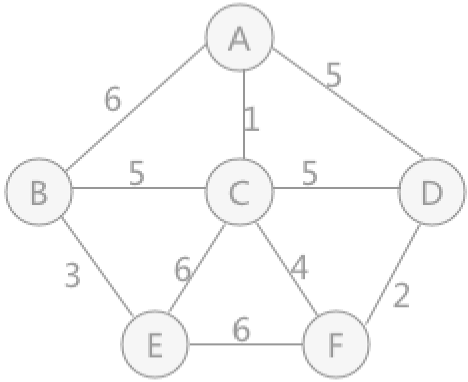
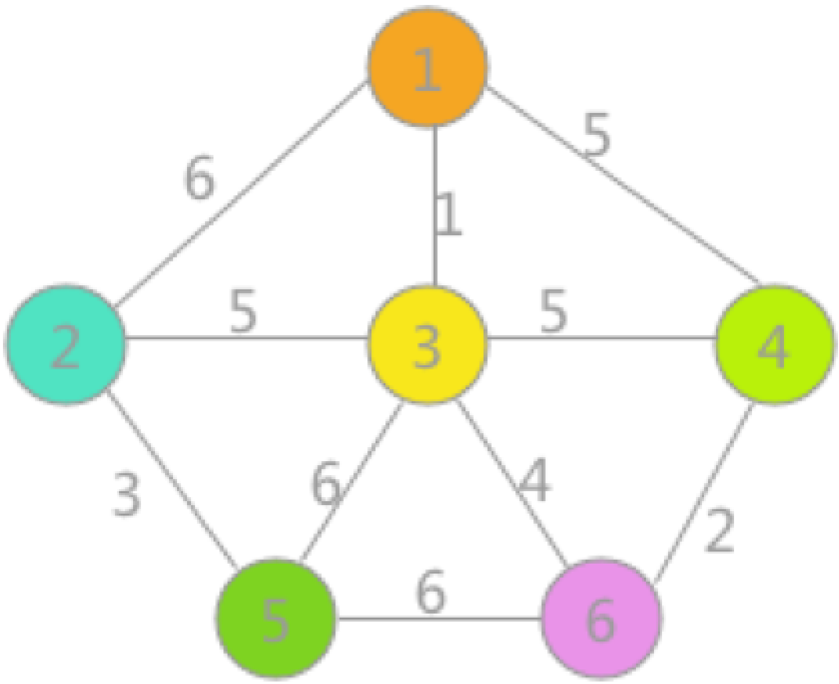


图 1 连通网

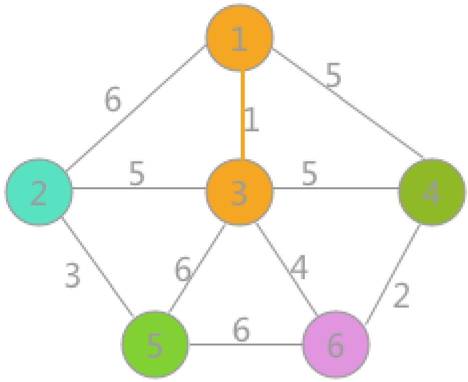
例如，使用克鲁斯卡尔算法找图 1 的最小生成树的过程为：

首先，在初始状态下，对各顶点赋予不同的标记（用颜色区别），如下图所示：



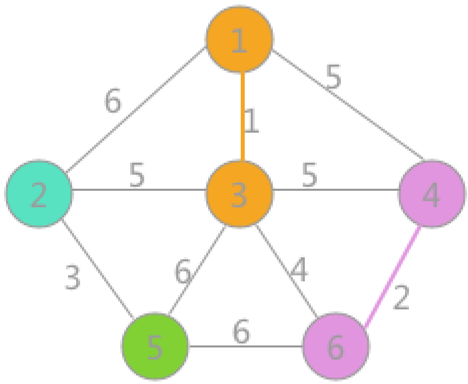
(1)

对所有边按照权值的大小进行排序，按照从小到大的顺序进行判断，首先是（1，3），由于顶点 1 和顶点 3 标记不同，所以可以构成生成树的一部分，遍历所有顶点，将与顶点 3 标记相同的全部更改为顶点 1 的标记，如（2）所示：



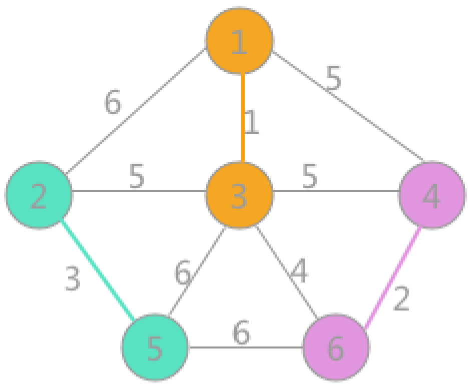
(2)

其次是（4，6）边，两顶点标记不同，所以可以构成生成树的一部分，更新所有顶点的标记为：



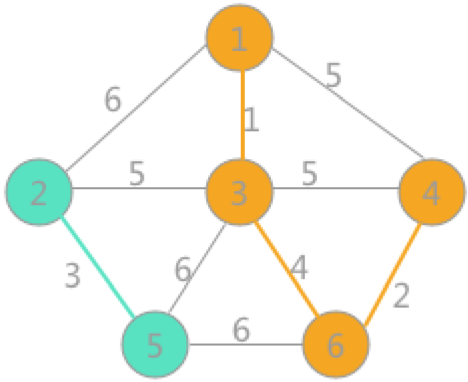
(3)

其次是（2，5）边，两顶点标记不同，可以构成生成树的一部分，更新所有顶点的标记为：



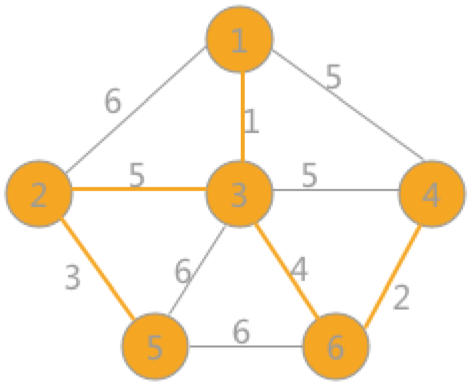
(4)

然后最小的是 3,6 边，两者标记不同，可以连接，遍历所有顶点，将与顶点 6 标记相同的所有顶点的标记更改为顶点 1 的标记：



（5）

继续选择权值最小的边，此时会发现，权值为 5 的边有 3 个，其中（1，4）和（3，4）各自两顶点的标记一样，如果连接会产生回路，所以舍去，而（2，3）标记不一样，可以选择，将所有与顶点 2 标记相同的顶点的标记全部改为同顶点 3 相同的标记：



（6）

当选取的边的数量相比与顶点的数量小 1 时，说明最小生成树已经生成。所以最终采用克鲁斯卡尔算法得到的最小生成树为（6）所示。

最小生成树

```
#include <bits/stdc++.h>
using namespace std;

const int N = 2e5 + 10;

int p[N]; //并查集数组，p[i]存储i的祖宗节点

struct Edge
{
```

```
    int u, v, w;

    bool operator<(const Edge &rhs) const
    {
        return w < rhs.w;
    }
} e[N];

int find(int x) //并查集查找x的祖宗节点
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int main()
{
    int n, m, u, v, w, ans = 0, cnt = 0; //cnt表示已加入最小生成树的边的个数
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        cin >> u >> v >> w;
        e[i] = {u, v, w};
    }
    sort(e, e + m); //对所有边权从小到大排序

    for (int i = 1; i <= n; i++) p[i] = i; //初始化并查集数组

    for (int i = 0; i < m; i++) //从小到大枚举所有边
    {
        u = e[i].u, v = e[i].v, w = e[i].w;
        u = find(u), v = find(v); //分别查找u和v的祖宗节点
        if (u != v) //两个点不在一个集合中
        {
            p[u] = v; //合并集合
            ans += w;
            cnt++;
        }
    }
```

```
    if (cnt < n - 1) cout << "impossible"; //边数不够，则不连通
    else cout << ans;
}
```